

Figure 3.1: The Sudoku example program for Android

Now let’s see how we can use this information to create the Sudoku opening screen.

# Creating the Opening Screen

Project name: Sudoku Build Target: Android 2.2 Application name: Sudoku

Package name: org.example.sudoku Create Activity: Sudoku

Min SDK Version: 8

**package** org.example.sudoku;

**import** android.app.Activity;

**import** android.os.Bundle;

**public class** Sudoku **extends** Activity {

/\*\* Called when the activity is first created. \*/

@Override

**public void** onCreate(Bundle savedInstanceState) { **super**.onCreate(savedInstanceState); setContentView(R.layout.main);

}

}

Android calls the onCreate( ) method of your activity to initialize it. The call to setContentView( ) fills in the contents of the activity’s screen with an Android view widget.

We could have used several lines of Java code, and possibly another class or two, to define the user interface procedurally. But instead, the plug-in chose the declarative route, and we’ll continue along those lines. In the previous code, R.layout.main is a resource identifier that refers to the main.xml file in the res/layout directory (see Figure 3.2, on the following page). main.xml declares the user interface in XML, so that’s the file we need to modify. At runtime, Android parses and instanti- ates (*inflates*) the resource defined there and sets it as the view for the current activity.

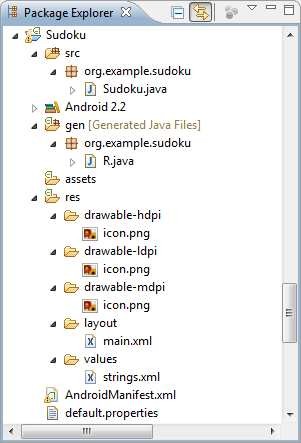


Figure 3.2: Initial resources in the Sudoku project

It’s important to note that the R class is managed automatically by the Android Eclipse plug-in. When you put a file anywhere in the res direc- tory, the plug-in notices the change and adds resource IDs in R.java in the gen directory for you. If you remove or change a resource file, R.java is kept in sync. If you bring up the file in the editor, it will look something like this:

Download **Sudokuv0/gen/org/example/sudoku/R.java**

/\* AUTO-GENERATED FILE. DO NOT MODIFY.

\*

* This class was automatically generated by the
* aapt tool from the resource data it found. It
* should not be modified by hand.

\*/

**package** org.example.sudoku;

**public final class** R {

**public static final class** attr {

}

**public static final class** drawable {

**public static final int** icon=0x7f020000;

}

**public static final class** layout {

**public static final int** main=0x7f030000;

}

**public static final class** string {

**public static final int** app\_name=0x7f040001;

**public static final int** hello=0x7f040000;

}

}

Next we see a reference to <LinearLayout>:

<LinearLayout xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:orientation="vertical" android:layout\_width="fill\_parent" android:layout\_height="fill\_parent" >

<!-- ... -->

**</LinearLayout>**

A layout is a container for one or more child objects and a behavior to position them on the screen within the rectangle of the parent object. Here is a list of the most common layouts provided by Android:

* + FrameLayout: Arranges its children so they all start at the top left of the screen. This is used for tabbed views and image switchers.
  + LinearLayout: Arranges its children in a single column or row. This is the most common layout you will use.
  + RelativeLayout: Arranges its children in relation to each other or to the parent. This is often used in forms.
  + TableLayout: Arranges its children in rows and columns, similar to an HTML table.

Some parameters are common to all layouts:

xmlns:andr[oid="http://schem](http://schemas.android.com/apk/res/android)as[.android.com](http://schemas.android.com/apk/res/android)/a[pk/res/andr](http://schemas.android.com/apk/res/android)o[id"](http://schemas.android.com/apk/res/android)

Defines the XML namespace for Android. You should define this once, on the first XML tag in the file.

android:layout\_width="fill\_parent", android:layout\_height="fill\_parent"

Takes up the entire width and height of the parent (in this case, the window). Possible values are fill\_parent and wrap\_content.

Inside the <LinearLayout> tag you’ll find one child widget:

<TextView

android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/hello" />

This defines a simple text label. Let’s replace that with some different text and a few buttons. Here’s our first attempt:

Download **Sudokuv1/res/layout/main1.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<LinearLayout xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:orientation="vertical" android:layout\_width="fill\_parent" android:layout\_height="fill\_parent" >

<TextView

android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/main\_title" />

<Button

android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/continue\_label" />

<Button

android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/new\_game\_label" />

<Button

android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/about\_label" />

<Button

android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/exit\_label" />

**</LinearLayout>**

If you see warnings in the editor about missing grammar constraints (DTD or XML schema), just ignore them. Instead of hard-coding English text into the layout file, we use the @string/*resid* syntax to refer to strings in the res/values/strings.xml file. You can have different versions of this and other resource files based on the locale or other parameters such as screen resolution and orientation.

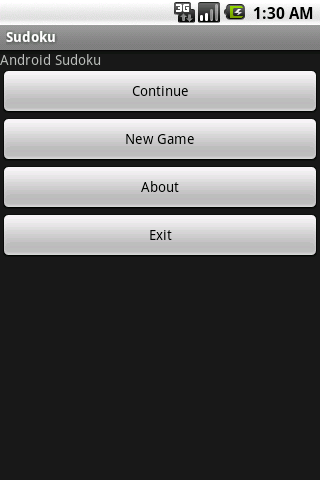


Figure 3.3: First version of the opening screen

Open that file now, switch to the strings.xml tab at the bottom if neces- sary, and enter this:

Download **Sudokuv1/res/values/strings.xml**

**<?xml version="1.0" encoding="utf-8"?>**

**<resources>**

**<string name="app\_name">**Sudoku**</string>**

**<string name="main\_title">**Android Sudoku**</string>**

**<string name="continue\_label">**Continue**</string>**

**<string name="new\_game\_label">**New Game**</string>**

**<string name="about\_label">**About**</string>**

**<string name="exit\_label">**Exit**</string>**

**</resources>**

Save strings.xml so Eclipse will rebuild the project. When you run the program now, you should see something like Figure 3.3.

Note: Because this is the third edition of the book, I have a pretty good idea where most people run into trouble. This is it, right here. You’ve made a lot of changes, so don’t be surprised if you get an error mes-

sage instead of the opening screen. Don’t panic; just skip ahead to Section 3.10, *Debugging*, on page 69 for advice on how to diagnose the problem. Usually a clue to the problem is waiting for you in the LogCat view. Sometimes selecting Project > Clean will fix things. If you’re still stuck, drop by the book’s web forum, and somebody would be happy to help you there.2

The current screen is readable, but it could use some cosmetic changes. Let’s make the title text larger and centered, make the buttons smaller, and use a different background color. Here’s the color definition, which you should put in res/values/colors.xml:

Download **Sudokuv1/res/values/colors.xml**

**<?xml version="1.0" encoding="utf-8"?>**

**<resources>**

**<color name="background">**#3500ffff**</color>**

**</resources>**

And here’s the new layout:

Download **Sudokuv1/res/layout/main.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<LinearLayout xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:background="@color/background" android:layout\_height="fill\_parent" android:layout\_width="fill\_parent"

android:padding="30dip" android:orientation="horizontal" >

<LinearLayout android:orientation="vertical" android:layout\_height="wrap\_content" android:layout\_width="fill\_parent" android:layout\_gravity="center" >

<TextView

android:text="@string/main\_title" android:layout\_height="wrap\_content" android:layout\_width="wrap\_content" android:layout\_gravity="center" android:layout\_marginBottom="25dip" android:textSize="24.5sp" />

<Button

android:id="@+id/continue\_button" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/continue\_label" />

1. <http://forums.pragprog.com/forums/152>

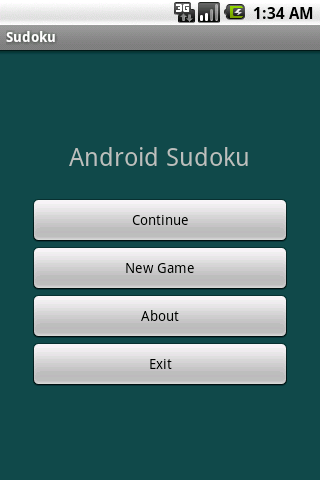


Figure 3.4: Opening screen with new layout

<Button

android:id="@+id/new\_button" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/new\_game\_label" />

<Button

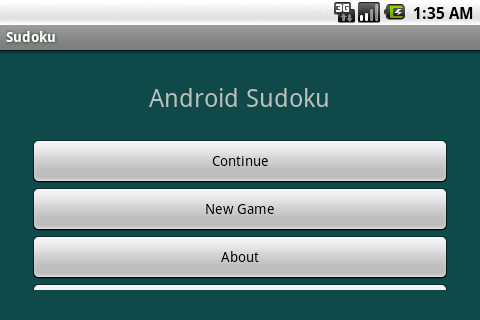
android:id="@+id/about\_button" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/about\_label" />

<Button

android:id="@+id/exit\_button" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:text="@string/exit\_label" />

**</LinearLayout>**

**</LinearLayout>**



# Using Alternate Resources

As a test, try switching the emulator to landscape mode ( Ctrl+F11 or the 7 or 9 key on the keypad). Oops! The Exit button runs off the bottom of the screen (see Figure 3.5). How do we fix that?

You could try to adjust the layout so that it works with all orienta- tions. Unfortunately, that’s often not possible or leads to odd-looking screens. When that happens, you’ll need to create a different layout for landscape mode. That’s the approach we’ll take here.

Create a file called res/layout-land/main.xml (note the -land suffix) that contains the following layout:

Download **Sudokuv1/res/layout-land/main.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<LinearLayout xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:background="@color/background" android:layout\_height="fill\_parent" android:layout\_width="fill\_parent"

android:padding="15dip" android:orientation="horizontal" >

<LinearLayout android:orientation="vertical" android:layout\_height="wrap\_content" android:layout\_width="fill\_parent" android:layout\_gravity="center" android:paddingLeft="20dip" android:paddingRight="20dip" >

<TextView

android:text="@string/main\_title" android:layout\_height="wrap\_content" android:layout\_width="wrap\_content" android:layout\_gravity="center" android:layout\_marginBottom="20dip" android:textSize="24.5sp" />

<TableLayout android:layout\_height="wrap\_content" android:layout\_width="wrap\_content" android:layout\_gravity="center" android:stretchColumns="\*" >

**<TableRow>**

<Button

android:id="@+id/continue\_button" android:text="@string/continue\_label" />

<Button

android:id="@+id/new\_button" android:text="@string/new\_game\_label" />

**</TableRow>**

**<TableRow>**

<Button

android:id="@+id/about\_button" android:text="@string/about\_label" />

<Button

android:id="@+id/exit\_button" android:text="@string/exit\_label" />

**</TableRow>**

**</TableLayout>**

**</LinearLayout>**

**</LinearLayout>**



Figure 3.6: Using a landscape-specific layout lets us see all the buttons.

This uses a TableLayout to create two columns of buttons. Now run the program again (see Figure 3.6). Even in landscape mode, all the buttons are visible.

You can use resource suffixes to specify alternate versions of any re- sources, not just the layout. For example, you can use them to provide localized text strings in different languages. Android’s screen density support depends heavily on these resource suffixes (see Section 13.5, *All Screens Great and Small*, on page 267).

# Implementing an About Box

When the user selects the About button, meaning that either they touch it (if they have a touch screen) or they navigate to it with the D-pad (directional pad) or trackball and press the selection button, we want to pop up a window with some information about Sudoku.

After scrolling through the text, the user can press the Back button to dismiss the window.

We can accomplish this in several ways:

* + - Define a new Activity, and start it.
    - Use the AlertDialog class, and show it.
    - Subclass Android’s Dialog class, and show that.

For this example, let’s define a new activity. Like the main Sudoku activ- ity, the About activity will need a layout file. We will name it res/layout/ about.xml:

Download **Sudokuv1/res/layout/about.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<ScrollView xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:layout\_width="fill\_parent" android:layout\_height="fill\_parent" android:padding="10dip" >

<TextView

android:id="@+id/about\_content" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@string/about\_text" />

**</ScrollView>**

We need only one version of this layout because it will look fine in both portrait and landscape modes.

Now add strings for the title of the About dialog box and the text it contains to res/values/strings.xml:

Download **Sudokuv1/res/values/strings.xml**

**<string name="about\_title">**About Android Sudoku**</string>**

**<string name="about\_text">**\

Sudoku is a logic-based number placement puzzle. Starting with a partially completed 9x9 grid, the objective is to fill the grid so that each

row, each column, and each of the 3x3 boxes (also called **<i>**blocks**</i>**) contains the digits

1 to 9 exactly once.

**</string>**

Note how a string resource can contain simple HTML formatting and can span multiple lines. In case you’re wondering, the backslash char- acter (\) in about\_text prevents an extra blank from appearing before the first word.

The About activity should be defined in About.java. All it needs to do is override onCreate( ) and call setContentView( ). To create a new class in Eclipse, use File > New > Class. Specify the following:

Source folder: Sudoku/src Package: org.example.sudoku Name: About

Edit the class so it looks like this:

Download **Sudokuv1/src/org/example/sudoku/About.java**

**package** org.example.sudoku;

**import** android.app.Activity;

**import** android.os.Bundle;

**public class** About **extends** Activity {

@Override

**protected void** onCreate(Bundle savedInstanceState) { **super**.onCreate(savedInstanceState); setContentView(R.layout.about);

}

}

Next we need to wire all this up to the About button in the Sudoku class. Start by adding a few imports that we’ll need to Sudoku.java:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**import** android.content.Intent;

**import** android.view.View;

**import** android.view.View.OnClickListener;

In the onCreate( ) method, add code to call findViewById( ) to look up an Android view given its resource ID, and add code to call setOnClickLis- tener( ) to tell Android which object to tickle when the user touches or clicks the view:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

@Override

**public void** onCreate(Bundle savedInstanceState) { **super**.onCreate(savedInstanceState); setContentView(R.layout.main);

// Set up click listeners for all the buttons

View continueButton = findViewById(R.id.continue\_button); continueButton.setOnClickListener(**this**);

View newButton = findViewById(R.id.new\_button); newButton.setOnClickListener(**this**);

View aboutButton = findViewById(R.id.about\_button); aboutButton.setOnClickListener(**this**);

View exitButton = findViewById(R.id.exit\_button); exitButton.setOnClickListener(**this**);

}

While we’re in here, we do the same for all the buttons. Recall that constants like R.id.about\_button are created by the Eclipse plug-in in R.java when it sees @+id/about\_button in res/layout/main.xml.

The setOnClickListener( ) method needs to be passed an object that imple- ments the OnClickListener Java interface. We’re passing it the this vari- able, so we had better make the current class (Sudoku) implement that interface, or we’ll get a compiler error. OnClickListener has one method in it called onClick( ), so we have to add that method to our class as well:3

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**public class** Sudoku **extends** Activity **implements** OnClickListener {

// ...

**public void** onClick(View v) {

**switch** (v.getId()) {

**case** R.id.about\_button:

Intent i = **new** Intent(**this**, About.**class**); startActivity(i);

**break**;

// More buttons go here (if any) ...

}

}

}

To start an activity in Android, we first need to create an instance of the Intent class. There are two kinds of intents: *public* (named) intents that are registered with the system and can be called from any appli- cation and *private* (anonymous) intents that are used within a single application. For this example, we just need the latter kind. If you run the program and select the About button now, you will get an error (see Figure 3.7, on the following page). What happened?

We forgot one important step: every activity needs to be declared in AndroidManifest.xml. To do that, double-click the file to open it, switch to XML mode if necessary by selecting the AndroidManifest.xml tab at the bottom, and add a new <activity> tag after the closing tag of the first one:

<activity android:name=".About" android:label="@string/about\_title" >

**</activity>**

Now if you save the manifest, run the program again, and select the About button, you should see something like Figure 3.8, on page 62. Press the Back button ( Esc on the emulator) when you’re done.

1. If you’re a Java expert, you may be wondering why we didn’t use an anonymous inner class to handle the clicks. You could, but according to the Android developers, every new inner class takes up an extra 1KB of memory.



Figure 3.7: Mountain View, we have a problem

That looks OK, but wouldn’t it be nice if we could see the initial screen behind the About text?

# Applying a Theme

A *theme* is a collection of styles that override the look and feel of Android widgets. Themes were inspired by Cascading Style Sheets (CSS) used for web pages—they separate the content of a screen and its presen- tation or style. Android is packaged with several themes that you can reference by name,4 or you can make up your own theme by subclass- ing existing ones and overriding their default values.

We could define our own custom theme in res/values/styles.xml, but for this example we’ll just take advantage of a predefined one. To use it, open the AndroidManifest.xml editor again, and change the definition of the About activity so it has a theme property.

1. See <http://d.android.com/reference/android/R.style.html> for symbols beginning with “Theme\_.”

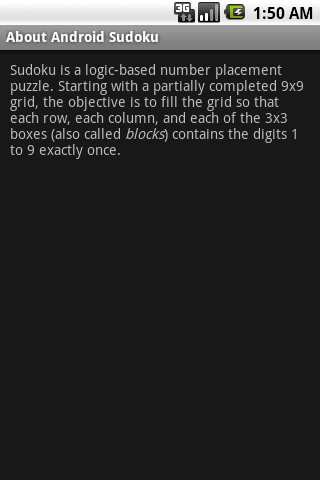


Figure 3.8: First version of the About screen

Download **Sudokuv1/AndroidManifest.xml**

<activity android:name=".About" android:label="@string/about\_title" android:theme="@android:style/Theme.Dialog" >

**</activity>**

The @android: prefix in front of the style name means this is a refer- ence to a resource defined by Android, not one that is defined in your program.

Running the program again, the About box now looks like Figure 3.9, on the following page.

Many programs need menus and options, so the next two sections will show you how to define them.

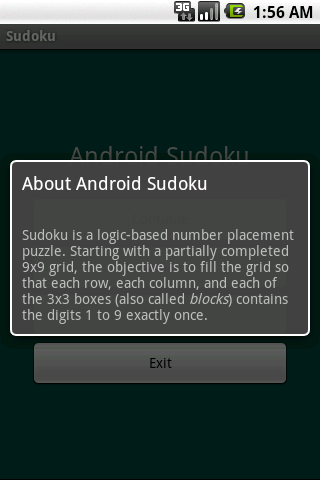


Figure 3.9: About screen after applying the dialog box theme

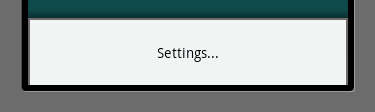


Figure 3.10: The options menu contains one item for changing the Set- tings

# Adding a Menu

Android supports two kinds of menus. First, there is the menu you get when you press the physical Menu button. Second, there is a context menu that pops up when you press and hold your finger on the screen (or press and hold the trackball or the D-pad center button).

Let’s do the first kind so that when the user presses the Menu key, they’ll open a menu like the one in Figure 3.10. First we need to define a few strings that we’ll use later:

Download **Sudokuv1/res/values/strings.xml**

**<string name="settings\_label">**Settings...**</string>**

**<string name="settings\_title">**Sudoku settings**</string>**

**<string name="settings\_shortcut">**s**</string>**

**<string name="music\_title">**Music**</string>**

**<string name="music\_summary">**Play background music**</string>**

**<string name="hints\_title">**Hints**</string>**

**<string name="hints\_summary">**Show hints during play**</string>**

Then we define the menu using XML in res/menu/menu.xml:

Download **Sudokuv1/res/menu/menu.xml**

**<?xml version="1.0" encoding="utf-8"?>**

**<menu** [**xmlns:android="http://schemas.android.com/apk/res/android">**](http://schemas.android.com/apk/res/android)

<item android:id="@+id/settings" android:title="@string/settings\_label" android:alphabeticShortcut="@string/settings\_shortcut" />

**</menu>**

Next we need to modify the Sudoku class to bring up the menu we just defined. To do that, we’ll need a few more imports:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**import** android.view.Menu;

**import** android.view.MenuInflater;

**import** android.view.MenuItem;

Then we override the Sudoku.onCreateOptionsMenu( ) method:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

@Override

**public boolean** onCreateOptionsMenu(Menu menu) { **super**.onCreateOptionsMenu(menu); MenuInflater inflater = getMenuInflater(); inflater.inflate(R.menu.menu, menu);

**return true**;

}

getMenuInflater( ) returns an instance of MenuInflater that we use to read the menu definition from XML and turns it into a real view. When the user selects any menu item, onOptionsItemSelected( ) will be called. Here’s the definition for that method:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

@Override

**public boolean** onOptionsItemSelected(MenuItem item) {

**switch** (item.getItemId()) {

**case** R.id.settings:

startActivity(**new** Intent(**this**, Prefs.**class**));

**return true**;

// More items go here (if any) ...

}

**return false**;

}

Prefs is a class that we’re going to define that displays all our preferences and allows the user to change them.

# Adding Settings

Android provides a nice facility for defining what all your program pref- erences are and how to display them using almost no code. You define the preferences in a resource file called res/xml/settings.xml:

Download **Sudokuv1/res/xml/settings.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<PreferenceScreen xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android)>

<CheckBoxPreference android:key="music" android:title="@string/music\_title"

android:summary="@string/music\_summary" android:defaultValue="true" />

<CheckBoxPreference android:key="hints" android:title="@string/hints\_title"

android:summary="@string/hints\_summary" android:defaultValue="true" />

**</PreferenceScreen>**

The Sudoku program has two settings: one for background music and one for displaying hints. The keys are constant strings that will be used under the covers in Android’s preferences database.

Next define the Prefs class, and make it extend PreferenceActivity:

Download **Sudokuv1/src/org/example/sudoku/Prefs.java**

**package** org.example.sudoku;

**import** android.os.Bundle;

**import** android.preference.PreferenceActivity;

**public class** Prefs **extends** PreferenceActivity {

@Override

**protected void** onCreate(Bundle savedInstanceState) { **super**.onCreate(savedInstanceState); addPreferencesFromResource(R.xml.settings);

}

}

The addPreferencesFromResource( ) method reads the settings definition from XML and inflates it into views in the current activity. All the heavy lifting takes place in the PreferenceActivity class.

Don’t forget to register the Prefs activity in AndroidManifest.xml:

Download **Sudokuv1/AndroidManifest.xml**

<activity android:name=".Prefs" android:label="@string/settings\_title" >

**</activity>**

Now rerun Sudoku, press the Menu key, select the Settings... item, and watch with amazement as the Sudoku settings page appears (see Fig- ure 3.11, on the next page). Try changing the values there and exiting the program, and then come back in and make sure they’re all still set.

Code that reads the settings and does something with them will be discussed in a different chapter (Chapter 6, *Storing Local Data*, on page 120). For now let’s move on to the New Game button.

# Starting a New Game

If you’ve played any Sudoku games, you know that some are easy and some are maddeningly hard. So when the user selects New Game, we want to pop up a dialog box asking them to select between three diffi- culty levels. Selecting from a list of things is easy to do in Android.

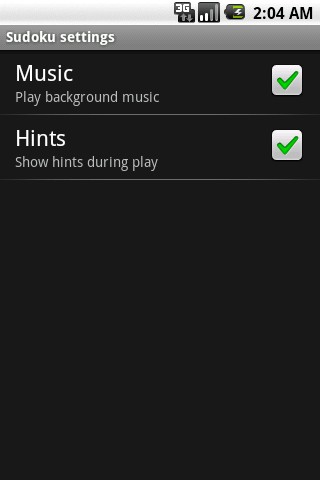


Figure 3.11: It’s not much to look at, but we got it for free.

First we’ll need a few more strings in res/values/strings.xml:

Download **Sudokuv1/res/values/strings.xml**

**<string name="new\_game\_title">**Difficulty**</string>**

**<string name="easy\_label">**Easy**</string>**

**<string name="medium\_label">**Medium**</string>**

**<string name="hard\_label">**Hard**</string>**

Create the list of difficulties as an array resource in res/values/arrays.xml:

Download **Sudokuv1/res/values/arrays.xml**

**<?xml version="1.0" encoding="utf-8"?>**

**<resources>**

**<array name="difficulty">**

**<item>**@string/easy\_label**</item>**

**<item>**@string/medium\_label**</item>**

**<item>**@string/hard\_label**</item>**

**</array>**

**</resources>**

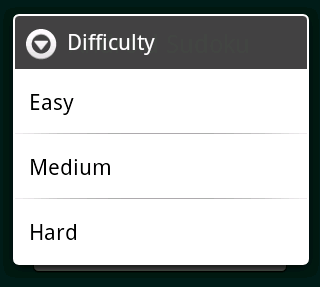


Figure 3.12: Difficulty selection dialog box

We’ll need a few more imports in the Sudoku class:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**import** android.app.AlertDialog;

**import** android.content.DialogInterface;

**import** android.util.Log;

Add code in the switch statement of the onClick( ) method to handle click- ing the New Game button:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**case** R.id.new\_button: openNewGameDialog(); **break**;

The openNewGameDialog( ) method takes care of creating the user inter- face for the difficulty list.

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**private static final** String TAG = "Sudoku" ;

**private void** openNewGameDialog() {

**new** AlertDialog.Builder(**this**)

.setTitle(R.string.new\_game\_title)

.setItems(R.array.difficulty,

**new** DialogInterface.OnClickListener() {

**public void** onClick(DialogInterface dialoginterface,

**int** i) { startGame(i);

}

})

.show();

}

**private void** startGame(**int** i) { Log.d(TAG, "clicked on " + i);

// Start game here...

}

The setItems( ) method takes two parameters: the resource ID of the item list and a listener that will be called when one of the items is selected.

When you run the program now and press New Game, you’ll get the dialog box in Figure 3.12, on the previous page.

We’re not actually going to start the game yet, so instead when you select a difficulty level, we just print a debug message using the Log.d( ) method, passing it a tag string and a message to print.

# Exiting the Game

This game doesn’t really need an Exit button, because the user can just press the Back key or the Home key to do something else. But I wanted to add one to show you how to terminate an activity.

Add this to the switch statement in the onClick( ) method:

Download **Sudokuv1/src/org/example/sudoku/Sudoku.java**

**case** R.id.exit\_button: finish();

**break**;

When the Exit button is selected, we call the finish( ) method. This shuts down the activity and returns control to the next activity on the Android application stack (usually the Home screen).

# Adding Graphics to Sudoku

It’s time to apply what we’ve learned to our Sudoku example. When we left it at the end of Chapter 3, the Sudoku game had an opening screen, an About dialog box, and a way to start a new game. But it was missing one very important part: the game! We’ll use the native 2D graphics library to implement that part.

## Starting the Game

First we need to fill in the code that starts the game. startGame( ) takes one parameter, the index of the difficulty name selected from the list.

Here’s the new definition:

Download **Sudokuv2/src/org/example/sudoku/Sudoku.java**

**private void** startGame(**int** i) { Log.d(TAG, "clicked on " + i);

Intent intent = **new** Intent(Sudoku.**this**, Game.**class**); intent.putExtra(Game.KEY\_DIFFICULTY, i); startActivity(intent);

}

**Sudoku Trivia**

A few years after it was published in the United States, Num- ber Place was picked up by the Japanese publisher Nikoli, who gave it the much cooler-sounding name Sudoku (which means “single number” in Japanese). From there it was exported around the world, and the rest is history. Sadly, Garns died in 1989 before getting a chance to see his creation become a worldwide sensation.

The game part of Sudoku will be another activity called Game, so we create a new intent to kick it off. We place the difficulty number in an extraData area provided in the intent, and then we call the startActivity( ) method to launch the new activity.

The extraData area is a map of key/value pairs that will be passed along to the intent. The keys are strings, and the values can be any prim- itive type, array of primitives, Bundle, or a subclass of Serializable or Parcelable.

## Defining the Game Class

Here’s the outline of the Game activity:

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**package** org.example.sudoku;

**import** android.app.Activity; **import** android.app.Dialog; **import** android.os.Bundle; **import** android.util.Log; **import** android.view.Gravity; **import** android.widget.Toast;

**public class** Game **extends** Activity {

**private static final** String TAG = "Sudoku" ;

**public static final** String KEY\_DIFFICULTY =

"org.example.sudoku.difficulty" ;

**public static final int** DIFFICULTY\_EASY = 0; **public static final int** DIFFICULTY\_MEDIUM = 1; **public static final int** DIFFICULTY\_HARD = 2;

**private int** puzzle[] = **new int**[9 \* 9];

**private** PuzzleView puzzleView;

@Override

**protected void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState); Log.d(TAG, "onCreate" );

**int** diff = getIntent().getIntExtra(KEY\_DIFFICULTY, DIFFICULTY\_EASY);

puzzle = getPuzzle(diff); calculateUsedTiles();

puzzleView = **new** PuzzleView(**this**); setContentView(puzzleView); puzzleView.requestFocus();

}

// ...

}

The onCreate( ) method fetches the difficulty number from the intent and selects a puzzle to play. Then it creates an instance of the PuzzleView class, setting the PuzzleView as the new contents of the view. Since this is a fully customized view, it was easier to do this in code than in XML.

The calculateUsedTiles( ) method, which is defined in Section 4.4, *The Rest of the Story*, on page 93, uses the rules of Sudoku to figure out, for each tile in the nine-by-nine grid, which numbers are not valid for the tile because they appear elsewhere in the horizontal or vertical direction or in the three-by-three subgrid.

This is an activity, so we need to register it in AndroidManifest.xml:

Download **Sudokuv2/AndroidManifest.xml**

<activity android:name=".Game" android:label="@string/game\_title" />

We also need to add a few more string resources to res/values/strings.xml:

Download **Sudokuv2/res/values/strings.xml**

**<string name="game\_title">**Game**</string>**

**<string name="no\_moves\_label">**No moves**</string>**

**<string name="keypad\_title">**Keypad**</string>**

## Defining the PuzzleView Class

Next we need to define the PuzzleView class. Instead of using an XML layout, this time let’s do it entirely in Java.

**What Size Is It Anyway?**

A common mistake made by new Android developers is to use the width and height of a view inside its constructor. When a view’s constructor is called, Android doesn’t know yet how big the view will be, so the sizes are set to zero. The real sizes are calculated during the layout stage, which occurs after construction but before anything is drawn. You can use the onSizeChanged( ) method to be notified of the values when they are known, or you can use the getWidth( ) and getHeight( ) meth- ods later, such as in the onDraw( ) method.

Here’s the outline:

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

**package** org.example.sudoku;

**import** android.content.Context; **import** android.graphics.Canvas; **import** android.graphics.Paint; **import** android.graphics.Rect;

**import** android.graphics.Paint.FontMetrics; **import** android.graphics.Paint.Style; **import** android.util.Log;

**import** android.view.KeyEvent; **import** android.view.MotionEvent; **import** android.view.View;

**import** android.view.animation.AnimationUtils;

**public class** PuzzleView **extends** View {

**private static final** String TAG = "Sudoku" ;

**private final** Game game;

**public** PuzzleView(Context context) {

**super**(context);

**this**.game = (Game) context; setFocusable(**true**); setFocusableInTouchMode(**true**);

}

// ...

}

In the constructor we keep a reference to the Game class and set the option to allow user input in the view. Inside PuzzleView, we need to implement the onSizeChanged( ) method. This is called after the view is created and Android knows how big everything is.

**Other Ways to Do It**

When I was writing this example, I tried several different ap- proaches such as using a button for each tile or declaring a grid of ImageView classes in XML. After many false starts, I found that the approach of having one view for the entire puzzle and drawing lines and numbers inside that proved to be the fastest and easiest way for this application.

It does have its drawbacks, though, such as the need to draw the selection and explicitly handle keyboard and touch events. When designing your own program, I recommend trying stan- dard widgets and views first and then falling back to custom drawing only if that doesn’t work for you.

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

**private float** width; // width of one tile **private float** height; // height of one tile **private int** selX; // X index of selection **private int** selY; // Y index of selection **private final** Rect selRect = **new** Rect();

@Override

**protected void** onSizeChanged(**int** w, **int** h, **int** oldw, **int** oldh) { width = w / 9f;

height = h / 9f; getRect(selX, selY, selRect);

Log.d(TAG, "onSizeChanged: width " + width + ", height "

+ height);

**super**.onSizeChanged(w, h, oldw, oldh);

}

**private void** getRect(**int** x, **int** y, Rect rect) { rect.set((**int**) (x \* width), (**int**) (y \* height), (**int**) (x

\* width + width), (**int**) (y \* height + height));

}

We use onSizeChanged( ) to calculate the size of each tile on the screen (1/9th of the total view width and height). Note this is a floating-point number, so it’s possible that we could end up with a fractional num- ber of pixels. selRect is a rectangle we’ll use later to keep track of the selection cursor.

At this point we’ve created a view for the puzzle, and we know how big it is. The next step is to draw the grid lines that separate the tiles on the board.

## Drawing the Board

Android calls a view’s onDraw( ) method every time any part of the view needs to be updated. To simplify things, onDraw( ) pretends that you’re re-creating the entire screen from scratch. In reality, you may be draw- ing only a small portion of the view as defined by the canvas’s clip rectangle. Android takes care of doing the clipping for you.

Start by defining a few new colors to play with in res/values/colors.xml:

Download **Sudokuv2/res/values/colors.xml**

**<color name="puzzle\_background">**#ffe6f0ff**</color>**

**<color name="puzzle\_hilite">**#ffffffff**</color>**

**<color name="puzzle\_light">**#64c6d4ef**</color>**

**<color name="puzzle\_dark">**#6456648f**</color>**

**<color name="puzzle\_foreground">**#ff000000**</color>**

**<color name="puzzle\_hint\_0">**#64ff0000**</color>**

**<color name="puzzle\_hint\_1">**#6400ff80**</color>**

**<color name="puzzle\_hint\_2">**#2000ff80**</color>**

**<color name="puzzle\_selected">**#64ff8000**</color>**

Here’s the basic outline for onDraw( ):

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

@Override

**protected void** onDraw(Canvas canvas) {

// Draw the background... Paint background = **new** Paint();

background.setColor(getResources().getColor( R.color.puzzle\_background));

canvas.drawRect(0, 0, getWidth(), getHeight(), background);

// Draw the board...

// Draw the numbers...

// Draw the hints...

// Draw the selection...

}

The first parameter is the Canvas on which to draw. In this code, we’re just drawing a background for the puzzle using the puzzle\_background color.

Now let’s add the code to draw the grid lines for the board:

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

// Draw the board...

// Define colors for the grid lines Paint dark = **new** Paint();

dark.setColor(getResources().getColor(R.color.puzzle\_dark));

Paint hilite = **new** Paint(); hilite.setColor(getResources().getColor(R.color.puzzle\_hilite));

Paint light = **new** Paint(); light.setColor(getResources().getColor(R.color.puzzle\_light));

// Draw the minor grid lines

**for** (**int** i = 0; i < 9; i++) {

canvas.drawLine(0, i \* height, getWidth(), i \* height, light);

canvas.drawLine(0, i \* height + 1, getWidth(), i \* height

+ 1, hilite);

canvas.drawLine(i \* width, 0, i \* width, getHeight(), light);

canvas.drawLine(i \* width + 1, 0, i \* width + 1, getHeight(), hilite);

}

// Draw the major grid lines

**for** (**int** i = 0; i < 9; i++) {

**if** (i % 3 != 0)

**continue**;

canvas.drawLine(0, i \* height, getWidth(), i \* height, dark);

canvas.drawLine(0, i \* height + 1, getWidth(), i \* height

+ 1, hilite);

canvas.drawLine(i \* width, 0, i \* width, getHeight(), dark); canvas.drawLine(i \* width + 1, 0, i \* width + 1,

getHeight(), hilite);

}

The code uses three different colors for the grid lines: a light color between each tile, a dark color between the three-by-three blocks, and a highlight color drawn on the edge of each tile to make them look like they have a little depth. The order in which the lines are drawn is important, since lines drawn later will be drawn over the top of earlier lines. You can see what this will look like in Figure 4.3, on the following page. Next, we need some numbers to go inside those lines.

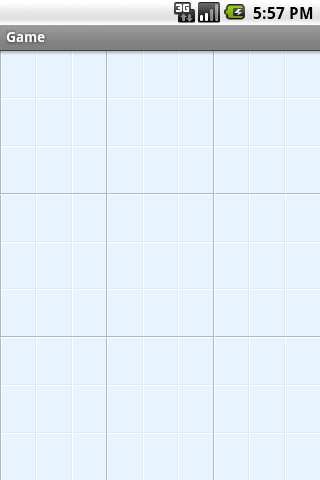


Figure 4.3: Drawing the grid lines with three shades of gray for an embossed effect

## Drawing the Numbers

The following code draws the puzzle numbers on top of the tiles. The tricky part here is getting each number positioned and sized so it goes in the exact center of its tile.

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

// Draw the numbers...

// Define color and style for numbers

Paint foreground = **new** Paint(Paint.ANTI\_ALIAS\_FLAG); foreground.setColor(getResources().getColor(

R.color.puzzle\_foreground)); foreground.setStyle(Style.FILL);

foreground.setTextSize(height \* 0.75f); foreground.setTextScaleX(width / height); foreground.setTextAlign(Paint.Align.CENTER);



Figure 4.4: Centering the numbers inside the tiles

// Draw the number in the center of the tile FontMetrics fm = foreground.getFontMetrics();

// Centering in X: use alignment (and X at midpoint)

**float** x = width / 2;

// Centering in Y: measure ascent/descent first **float** y = height / 2 - (fm.ascent + fm.descent) / 2; **for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) { canvas.drawText(**this**.game.getTileString(i, j), i

\* width + x, j \* height + y, foreground);

}

}

We call the getTileString( ) method (defined in Section 4.4, *The Rest of the Story*, on page 93) to find out what numbers to display. To calculate the size of the numbers, we set the font height to three-fourths the height of the tile, and we set the aspect ratio to be the same as the tile’s aspect

ratio. We can’t use absolute pixel or point sizes because we want the program to work at any resolution.

To determine the position of each number, we center it in both the x and y dimensions. The x direction is easy—just divide the tile width by 2. But for the y direction, we have to adjust the starting position downward a little so that the midpoint of the tile will be the midpoint of the number instead of its baseline. We use the graphics library’s FontMetrics class to tell how much vertical space the letter will take in total, and then we divide that in half to get the adjustment. You can see the results in Figure 4.4, on the preceding page.

That takes care of displaying the puzzle’s starting numbers (the givens). The next step is to allow the player to enter their guesses for all the blank spaces.

# Handling Input

One difference in Android programming—as opposed to, say, iPhone programming—is that Android phones come in many shapes and sizes and have a variety of input methods. They might have a keyboard, a D-pad, a touch screen, a trackball, or some combination of these.

A good Android program, therefore, needs to be ready to support what- ever input hardware is available, just like it needs to be ready to support any screen resolution.

## Defining and Updating the Selection

First we’re going to implement a little cursor that shows the player which tile is currently selected. The selected tile is the one that will be modified when the player enters a number. This code will draw the selection in onDraw( ):

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

// Draw the selection... Log.d(TAG, "selRect=" + selRect); Paint selected = **new** Paint();

selected.setColor(getResources().getColor( R.color.puzzle\_selected));

canvas.drawRect(selRect, selected);

We use the selection rectangle calculated earlier in onSizeChanged( ) to draw an alpha-blended color on top of the selected tile.

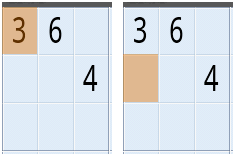


Figure 4.5: Drawing and moving the selection

Next we provide a way to move the selection by overriding the onKey- Down( ) method:

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

@Override

**public boolean** onKeyDown(**int** keyCode, KeyEvent event) { Log.d(TAG, "onKeyDown: keycode=" + keyCode + ", event="

+ event);

**switch** (keyCode) {

**case** KeyEvent.KEYCODE\_DPAD\_UP: select(selX, selY - 1); **break**;

**case** KeyEvent.KEYCODE\_DPAD\_DOWN: select(selX, selY + 1); **break**;

**case** KeyEvent.KEYCODE\_DPAD\_LEFT: select(selX - 1, selY); **break**;

**case** KeyEvent.KEYCODE\_DPAD\_RIGHT: select(selX + 1, selY); **break**;

default:

**return super**.onKeyDown(keyCode, event);

}

**return true**;

}

If the user has a directional pad (D-pad) and they press the up, down, left, or right button, we call select( ) to move the selection cursor in that direction.

How about a trackball? We could override the onTrackballEvent( ) method, but it turns out that if you don’t handle trackball events, Android will translate them into D-pad events automatically. Therefore, we can leave it out for this example.

Inside the select( ) method, we calculate the new x and y coordinates of the selection and then use getRect( ) again to calculate the new selection rectangle.

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

**private void** select(**int** x, **int** y) { invalidate(selRect);

selX = Math.min(Math.max(x, 0), 8);

selY = Math.min(Math.max(y, 0), 8); getRect(selX, selY, selRect); invalidate(selRect);

}

Notice the two calls to invalidate( ). The first one tells Android that the area covered by the old selection rectangle (on the left of Figure 4.5, on the previous page) needs to be redrawn. The second invalidate( ) call says that the new selection area (on the right of the figure) needs to be redrawn too. We don’t actually draw anything here.

This is an important point: never call any drawing functions except in the onDraw( ) method. Instead, you use the invalidate( ) method to mark rectangles as *dirty*. The window manager will combine all the dirty rect- angles at some point in the future and call onDraw( ) again for you. The dirty rectangles become the clip region, so screen updates are optimized to only those areas that change.

Now let’s provide a way for the player to enter a new number on the selected tile.

## Entering Numbers

To handle keyboard input, we just add a few more cases to the onKey- Down( ) method for the numbers 0 through 9 (0 or space means erase the number).

**Optimizing Refreshes**

In an earlier version of this example, I invalidated the entire screen whenever the cursor was moved. Thus, on every key press, the whole puzzle had to be redrawn. This caused it to lag noticeably. Switching the code to invalidate only the smallest rectangles that changed made it run much faster.

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

**case** KeyEvent.KEYCODE\_0:

**case** KeyEvent.KEYCODE\_SPACE: setSelectedTile(0); **break**; **case** KeyEvent.KEYCODE\_1: setSelectedTile(1); **break**; **case** KeyEvent.KEYCODE\_2: setSelectedTile(2); **break**; **case** KeyEvent.KEYCODE\_3: setSelectedTile(3); **break**; **case** KeyEvent.KEYCODE\_4: setSelectedTile(4); **break**; **case** KeyEvent.KEYCODE\_5: setSelectedTile(5); **break**; **case** KeyEvent.KEYCODE\_6: setSelectedTile(6); **break**; **case** KeyEvent.KEYCODE\_7: setSelectedTile(7); **break**; **case** KeyEvent.KEYCODE\_8: setSelectedTile(8); **break**; **case** KeyEvent.KEYCODE\_9: setSelectedTile(9); **break**; **case** KeyEvent.KEYCODE\_ENTER:

**case** KeyEvent.KEYCODE\_DPAD\_CENTER: game.showKeypadOrError(selX, selY); **break**;

To support the D-pad, we check for the Enter or center D-pad button in onKeyDown( ) and have it pop up a keypad that lets the user select which number to place.

For touch, we override the onTouchEvent( ) method and show the same keypad, which will be defined later:

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

@Override

**public boolean** onTouchEvent(MotionEvent event) {

**if** (event.getAction() != MotionEvent.ACTION\_DOWN)

**return super**.onTouchEvent(event);

select((**int**) (event.getX() / width), (**int**) (event.getY() / height));

game.showKeypadOrError(selX, selY);

Log.d(TAG, "onTouchEvent: x " + selX + ", y " + selY);

**return true**;

}

Ultimately, all roads will lead back to a call to setSelectedTile( ) to change the number on a tile:

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

**public void** setSelectedTile(**int** tile) {

**if** (game.setTileIfValid(selX, selY, tile)) { invalidate();// may change hints

} **else** {

// Number is not valid for this tile

Log.d(TAG, "setSelectedTile: invalid: " + tile);

}

}

The showKeypadOrError( ) and setTileIfValid( ) methods will be defined in Section 4.4, *The Rest of the Story*, on page 93.

Note the call to invalidate( ) with no parameters. That marks the whole screen as dirty, which violates my own advice earlier! However, in this case, it’s necessary because any new numbers added or removed might change the hints that we are about to implement in the next section.

## Adding Hints

How can we help the player out a little without solving the whole puzzle for them? How about if we draw the background of each tile differently depending on how many possible moves it has. Add this to onDraw( ) before drawing the selection:

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

// Draw the hints...

// Pick a hint color based on #moves left Paint hint = **new** Paint();

**int** c[] = { getResources().getColor(R.color.puzzle\_hint\_0), getResources().getColor(R.color.puzzle\_hint\_1), getResources().getColor(R.color.puzzle\_hint\_2), };

Rect r = **new** Rect();

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

**int** movesleft = 9 - game.getUsedTiles(i, j).length;

**if** (movesleft < c.length) { getRect(i, j, r); hint.setColor(c[movesleft]); canvas.drawRect(r, hint);

}

}

}

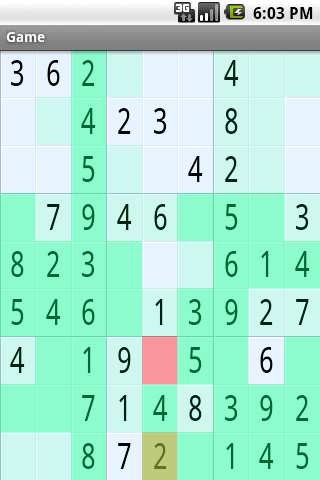


Figure 4.6: Tiles are highlighted based on how many possible values the tile can have.

We use three states for zero, one, and two possible moves. If there are zero moves, that means the player has done something wrong and needs to backtrack.

The result will look like Figure 4.6. Can you spot the mistake(s) made by the player?2

## Shaking Things Up

What if the user tries to enter an obviously invalid number, such as a number that already appears in the three-by-three block? Just for fun, let’s make the screen wiggle back and forth when they do that. First we add a call to the invalid number case in setSelectedTile( ).

1. The two numbers on the bottom row’s middle block are wrong.

Download **Sudokuv2/src/org/example/sudoku/PuzzleView.java**

Log.d(TAG, "setSelectedTile: invalid: " + tile); startAnimation(AnimationUtils.loadAnimation(game,

R.anim.shake));

This loads and runs a resource called R.anim.shake, defined in res/anim/ shake.xml, that shakes the screen for 1,000 milliseconds (1 second) by 10 pixels from side to side.

Download **Sudokuv2/res/anim/shake.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<translate

xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:fromXDelta="0"

android:toXDelta="10" android:duration="1000" android:interpolator="@anim/cycle\_7" />

The number of times to run the animation and the velocity and accel- eration of the animation are controlled by an animation interpolator defined in XML.

Download **Sudokuv2/res/anim/cycle\_7.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<cycleInterpolator xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:cycles="7" />

This particular one will cause the animation to be repeated seven times.

# The Rest of the Story

Now let’s go back and tie up a few loose ends, starting with the Key- pad class. These pieces are necessary for the program to compile and operate but have nothing to do with graphics. Feel free to skip ahead to Section 4.5, *Making More Improvements*, on page 103 if you like.

## Creating the Keypad

The keypad is handy for phones that don’t have keyboards. It displays a grid of the numbers 1 through 9 in an activity that appears on top of the puzzle. The whole purpose of the keypad dialog box is to return a number selected by the player.

Here’s the user interface layout from res/layout/keypad.xml:

Download **Sudokuv2/res/layout/keypad.xml**

**<?xml version="1.0" encoding="utf-8"?>**

<TableLayout xmlns:android=["http://schemas.android.com/apk/res/android"](http://schemas.android.com/apk/res/android) android:id="@+id/keypad"

android:orientation="vertical" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:stretchColumns="\*" >

**<TableRow>**

<Button android:id="@+id/keypad\_1" android:text="1" >

**</Button>**

<Button android:id="@+id/keypad\_2" android:text="2" >

**</Button>**

<Button android:id="@+id/keypad\_3" android:text="3" >

**</Button>**

**</TableRow>**

**<TableRow>**

<Button android:id="@+id/keypad\_4" android:text="4" >

**</Button>**

<Button android:id="@+id/keypad\_5" android:text="5" >

**</Button>**

<Button android:id="@+id/keypad\_6" android:text="6" >

**</Button>**

**</TableRow>**

**<TableRow>**

<Button android:id="@+id/keypad\_7" android:text="7" >

**</Button>**

<Button android:id="@+id/keypad\_8" android:text="8" >

**</Button>**

<Button android:id="@+id/keypad\_9" android:text="9" >

**</Button>**

**</TableRow>**

**</TableLayout>**

Next let’s define the Keypad class.

Here’s the outline:

Download **Sudokuv2/src/org/example/sudoku/Keypad.java**

**package** org.example.sudoku;

**import** android.app.Dialog; **import** android.content.Context; **import** android.os.Bundle; **import** android.view.KeyEvent; **import** android.view.View;

**public class** Keypad **extends** Dialog {

**protected static final** String TAG = "Sudoku" ;

**private final** View keys[] = **new** View[9];

**private** View keypad;

**private final int** useds[];

**private final** PuzzleView puzzleView;

**public** Keypad(Context context, **int** useds[], PuzzleView puzzleView) {

**super**(context); **this**.useds = useds;

**this**.puzzleView = puzzleView;

}

@Override

**protected void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setTitle(R.string.keypad\_title); setContentView(R.layout.keypad); findViews();

**for** (**int** element : useds) {

**if** (element != 0)

keys[element - 1].setVisibility(View.INVISIBLE);

}

setListeners();

}

// ...

}

If a particular number is not valid (for example, the same number already appears in that row), then we make the number invisible in the grid so the player can’t select it (see Figure 4.7, on the next page).



Figure 4.7: Invalid values are hidden in the keypad view.

The findViews( ) method fetches and saves the views for all the keypad keys and the main keypad window:

Download **Sudokuv2/src/org/example/sudoku/Keypad.java**

**private void** findViews() {

keypad = findViewById(R.id.keypad); keys[0] = findViewById(R.id.keypad\_1); keys[1] = findViewById(R.id.keypad\_2); keys[2] = findViewById(R.id.keypad\_3); keys[3] = findViewById(R.id.keypad\_4); keys[4] = findViewById(R.id.keypad\_5); keys[5] = findViewById(R.id.keypad\_6); keys[6] = findViewById(R.id.keypad\_7); keys[7] = findViewById(R.id.keypad\_8); keys[8] = findViewById(R.id.keypad\_9);

}

setListeners( ) loops through all the keypad keys and sets a listener for each one. It also sets a listener for the main keypad window.

Download **Sudokuv2/src/org/example/sudoku/Keypad.java**

**private void** setListeners() {

**for** (**int** i = 0; i < keys.length; i++) {

**final int** t = i + 1;

keys[i].setOnClickListener(**new** View.OnClickListener(){

**public void** onClick(View v) { returnResult(t);

}});

}

keypad.setOnClickListener(**new** View.OnClickListener(){

**public void** onClick(View v) { returnResult(0);

}});

}

When the player selects one of the buttons on the keypad, it calls the returnResult( ) method with the number for that button. If the player selects a place that doesn’t have a button, then returnResult( ) is called with a zero, indicating the tile should be erased.

onKeyDown( ) is called when the player uses the keyboard to enter a number:

Download **Sudokuv2/src/org/example/sudoku/Keypad.java**

@Override

**public boolean** onKeyDown(**int** keyCode, KeyEvent event) {

**int** tile = 0;

**switch** (keyCode) {

**case** KeyEvent.KEYCODE\_0:

**case** KeyEvent.KEYCODE\_SPACE: tile = 0; **break**; **case** KeyEvent.KEYCODE\_1: tile = 1; **break**; **case** KeyEvent.KEYCODE\_2: tile = 2; **break**; **case** KeyEvent.KEYCODE\_3: tile = 3; **break**; **case** KeyEvent.KEYCODE\_4: tile = 4; **break**; **case** KeyEvent.KEYCODE\_5: tile = 5; **break**; **case** KeyEvent.KEYCODE\_6: tile = 6; **break**; **case** KeyEvent.KEYCODE\_7: tile = 7; **break**; **case** KeyEvent.KEYCODE\_8: tile = 8; **break**; **case** KeyEvent.KEYCODE\_9: tile = 9; **break**; default:

**return super**.onKeyDown(keyCode, event);

}

**if** (isValid(tile)) { returnResult(tile);

}

**return true**;

}

If the number is valid for the current tile, then it calls returnResult( ); otherwise, the keystroke is ignored.

The isValid( ) method checks to see whether the given number is valid for the current position:

Download **Sudokuv2/src/org/example/sudoku/Keypad.java**

**private boolean** isValid(**int** tile) {

**for** (**int** t : useds) {

**if** (tile == t)

**return false**;

}

**return true**;

}

If it appears in the used array, then it’s not valid because the same number is already used in the current row, column, or block.

The returnResult( ) method is called to return the number selected to the calling activity:

Download **Sudokuv2/src/org/example/sudoku/Keypad.java**

**private void** returnResult(**int** tile) { puzzleView.setSelectedTile(tile); dismiss();

}

We call the PuzzleView.setSelectedTile() method to change the puzzle’s cur- rent tile. The dismiss call terminates the Keypad dialog box. Now that we have the activity, let’s call it in the Game class and retrieve the result:

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**protected void** showKeypadOrError(**int** x, **int** y) {

**int** tiles[] = getUsedTiles(x, y);

**if** (tiles.length == 9) {

Toast toast = Toast.makeText(**this**, R.string.no\_moves\_label, Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0); toast.show();

} **else** {

Log.d(TAG, "showKeypad: used=" + toPuzzleString(tiles)); Dialog v = **new** Keypad(**this**, tiles, puzzleView); v.show();

}

}

To decide which numbers are possible, we pass the Keypad a string in the extraData area containing all the numbers that have already been used.

## Implementing the Game Logic

The rest of the code in Game.java concerns itself with the logic of the game, in particular with determining which are and aren’t valid moves according to the rules. The setTileIfValid( ) method is a key part of that. Given an x and y position and the new value of a tile, it changes the tile only if the value provided is valid.

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**protected boolean** setTileIfValid(**int** x, **int** y, **int** value) {

**int** tiles[] = getUsedTiles(x, y);

**if** (value != 0) {

**for** (**int** tile : tiles) {

**if** (tile == value)

**return false**;

}

}

setTile(x, y, value); calculateUsedTiles(); **return true**;

}

To detect valid moves, we create an array for every tile in the grid. For each position, it keeps a list of filled-in tiles that are currently visible from that position. If a number appears on the list, then it won’t be valid for the current tile. The getUsedTiles( ) method retrieves that list for a given tile position:

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**private final int** used[][][] = **new int**[9][9][];

**protected int**[] getUsedTiles(**int** x, **int** y) {

**return** used[x][y];

}

The array of used tiles is somewhat expensive to compute, so we cache the array and recalculate it only when necessary by calling calculate- UsedTiles( ):

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**private void** calculateUsedTiles() {

**for** (**int** x = 0; x < 9; x++) {

**for** (**int** y = 0; y < 9; y++) {

used[x][y] = calculateUsedTiles(x, y);

// Log.d(TAG, "used[" + x + "][" + y + "] = "

// + toPuzzleString(used[x][y]));

}

}

}

calculateUsedTiles( ) simply calls calculateUsedTiles(x, y) on every position in the nine-by-nine grid:

Download **Sudokuv2/src/org/example/sudoku/Game.java**

Line 1 **private int**[] calculateUsedTiles(**int** x, **int** y) {

* **int** c[] = **new int**[9];
* // horizontal

- **for** (**int** i = 0; i < 9; i++) {

5 **if** (i == y)

* **continue**;
* **int** t = getTile(x, i);

- **if** (t != 0)

- c[t - 1] = t;

10 }

* // vertical

- **for** (**int** i = 0; i < 9; i++) {

* **if** (i == x)
* **continue**;

15 **int** t = getTile(i, y);

- **if** (t != 0)

- c[t - 1] = t;

- }

* // same cell block

20 **int** startx = (x / 3) \* 3;

* **int** starty = (y / 3) \* 3;
* **for** (**int** i = startx; i < startx + 3; i++) {
* **for** (**int** j = starty; j < starty + 3; j++) {

- **if** (i == x && j == y)

25 **continue**;

* **int** t = getTile(i, j);

- **if** (t != 0)

- c[t - 1] = t;

- }

30 }

* // compress
* **int** nused = 0;
* **for** (**int** t : c) {

- **if** (t != 0)

35 nused++;

- }

* **int** c1[] = **new int**[nused];
* nused = 0;
* **for** (**int** t : c) {

40 **if** (t != 0)

* c1[nused++] = t;

- }

* **return** c1;

- }

We start with an array of nine zeros. On line 4, we check all the tiles on the same horizontal row as the current tile, and if a tile is occupied, we stuff its number into the array.

On line 12, we do the same thing for all the tiles on the same vertical column, and on line 20, we do the same for tiles in the three-by-three block.

The last step, starting at line 32, is to compress the zeros out of the array before we return it. We do this so that array.length can be used to quickly tell how many used tiles are visible from the current position.

## Miscellaneous

Here are a few other utility functions and variables that round out the implementation. easyPuzzle, mediumPuzzle, and hardPuzzle are our hard- coded Sudoku puzzles for easy, medium, and hard difficulty levels, respectively.

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**private final** String easyPuzzle =

"360000000004230800000004200" +

"070460003820000014500013020" +

"001900000007048300000000045" ;

**private final** String mediumPuzzle =

"650000070000506000014000005" +

"007009000002314700000700800" +

"500000630000201000030000097" ;

**private final** String hardPuzzle =

"009000000080605020501078000" +

"000000700706040102004000000" +

"000720903090301080000000600" ;

getPuzzle( ) simply takes a difficulty level and returns a puzzle:

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**private int**[] getPuzzle(**int** diff) { String puz;

// TODO: Continue last game

**switch** (diff) {

**case** DIFFICULTY\_HARD:

puz = hardPuzzle;

**break**;

**case** DIFFICULTY\_MEDIUM:

puz = mediumPuzzle;

**break**;

**case** DIFFICULTY\_EASY:

default:

puz = easyPuzzle;

**break**;

}

**return** fromPuzzleString(puz);

}

Later we’ll change getPuzzle( ) to implement a continue function.

toPuzzleString( ) converts a puzzle from an array of integers to a string.

fromPuzzleString( ) does the opposite.

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**static private** String toPuzzleString(**int**[] puz) { StringBuilder buf = **new** StringBuilder();

**for** (**int** element : puz) { buf.append(element);

}

**return** buf.toString();

}

**static protected int**[] fromPuzzleString(String string) {

**int**[] puz = **new int**[string.length()]; **for** (**int** i = 0; i < puz.length; i++) {

puz[i] = string.charAt(i) - '0' ;

}

**return** puz;

}

The getTile( ) method takes x and y positions and returns the number currently occupying that tile. If it’s zero, that means the tile is blank.

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**private int** getTile(**int** x, **int** y) {

**return** puzzle[y \* 9 + x];

}

**private void** setTile(**int** x, **int** y, **int** value) { puzzle[y \* 9 + x] = value;

}

getTileString( ) is used when displaying a tile. It will return either a string with the value of the tile or an empty string if the tile is blank.

Download **Sudokuv2/src/org/example/sudoku/Game.java**

**protected** String getTileString(**int** x, **int** y) {

**int** v = getTile(x, y);

**if** (v == 0)

**return** "" ;

**else**

**return** String.valueOf(v);

}

Once all these pieces are in place, you should have a playable Sudoku game. Give it a try to verify it works. As with any code, though, there is room for improvement.

# Making More Improvements

Although the code presented in this chapter performs acceptably for a Sudoku game, more complex programs will likely need to be more carefully written in order to squeeze the last drop of performance out of the device. In particular, the onDraw( ) method is a very performance- critical piece of code, so it’s best to do as little as possible there.

Here are some ideas for speeding up this method:

* If possible, avoid doing any object allocations in the method

onDraw( ).

* Prefetch things such as color constants elsewhere (for example, in the view’s constructor).
* Create your Paint objects up front, and just use existing instances in onDraw( ).
* For values used multiple times, such as the width returned by getWidth( ), retrieve the value at the beginning of the method and then access it from your local copy.

As a further exercise for the reader, I encourage you to think about how you could make the Sudoku game graphically richer. For example, you could add some fireworks when the player solves the puzzle or make the tiles spin around like Vanna White does. A moving background behind the puzzle might be interesting. Let your imagination go wild. If you want to make a top-notch product, touches like this can add pizzazz to an otherwise ordinary offering.

# Adding Sounds to Sudoku

In this section, we’re going to take what we’ve learned and add back- ground music to the Sudoku game we’ve been building. One song will play during the opening screen, and another will play during the actual game. This will demonstrate not just how to play music but also some important life-cycle considerations.

To add music to the main screen, we just need to override these two methods in the Sudoku class:

Download **Sudokuv3/src/org/example/sudoku/Sudoku.java**

@Override

**protected void** onResume() { **super**.onResume(); Music.play(**this**, R.raw.main);

}

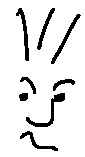
@Override

**protected void** onPause() { **super**.onPause(); Music.stop(**this**);

}

If you recall from Section 2.2, *It’s Alive!*, on page 35, the onResume( ) method is called when the activity is ready to begin interacting with the user. This is a good place to start up the music, so we put a Music.play( ) call there. The Music class will be defined shortly.

**Joe Asks. . .**



### Why Does It Restart the Video When I Rotate the Display?

Android assumes by default that your program knows nothing about screen rotations. To pick up possible resource changes, Android destroys and re-creates your activity from scratch. That means onCreate( ) is called again, which means the video is started again (as this example is currently written).

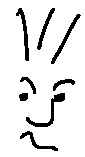
This behavior will be fine for 90 percent of all applications, so most developers will not have to worry about it. It’s even a useful way to test your application life-cycle and state- saving/restoring code (see Section 2.2, *It’s Alive!*, on page 35). However, there are a couple of ways to be smarter and opti- mize the transition.

The simplest way is to implement onRetainNonConfigurationIn- stance( ) in your activity to save some data that will be kept across the calls to onDestroy( ) and onCreate( ). When you come back, you use getLastNonConfigurationInstance( ) in the new instance of your activity to recover that information. You can keep anything, even references to your current intent and running threads.

The more complicated way is to use the android:configChanges= property in AndroidManifest.xml to inform Android which changes you can handle. For example, if you set it to *keyboardHidden*|*orientation*, then Android will not destroy and re-create your activity when the user flips the keyboard. Instead, it will call onConfigurationChanged(Configuration) and assume you know what you’re doing.∗

∗. See <http://d.android.com/reference/android/app/Activity.html#ConfigurationChanges>

for more details.



**Joe Asks. . .**

**Shouldn’t We Use a Background Service for Music?**

We haven’t said much about the Android Service class, but you may have seen it used in some music-playing examples on the Web. Basically, a Service is a way to start a background pro- cess that can run even after your current activity finishes. Ser- vices are similar to, but not quite the same as, Linux daemons. If you’re writing a general-purpose music player and want the music to continue while you’re reading mail or browsing the Web, then, yes, a Service would be appropriate. In most cases, though, you want the music to end when your program ends, so you don’t need to use the Service class.

R.raw.main refers to res/raw/main.mp3. You can find these sound files in the Sudokuv3 project of the downloadable samples on the book’s website.

The onPause( ) method is the paired bookend for onResume( ). Android pauses the current activity prior to resuming a new one, so in Sudoku, when you start a new game, the Sudoku activity will be paused, and then the Game activity will be started. onPause( ) will also be called when the user presses the Back or Home key. These are all places where we want our title music to stop, so we call Music.stop( ) in onPause( ).

Now let’s do something similar for the music on the Game activity:

Download **Sudokuv3/src/org/example/sudoku/Game.java**

@Override

**protected void** onResume() { **super**.onResume(); Music.play(**this**, R.raw.game);

}

@Override

**protected void** onPause() { **super**.onPause(); Music.stop(**this**);

}

**Sudoku Trivia**

Dozens of Sudoku variants exist, although none has gained the popularity of the original. One uses a sixteen-by-sixteen grid, with hexadecimal numbers. Another, called Gattai 5 or Samurai Sudoku, uses five nine-by-nine grids that overlap at the corner regions.

If you compare this to what we did to the Sudoku class, you’ll notice that we’re referencing a different sound resource, R.raw.game (res/raw/ game.mp3).

The final piece of the musical puzzle is the Music class, which will man- age the MediaPlayer class used to play the current music:

Download **Sudokuv3/src/org/example/sudoku/Music.java**

Line 1 **package** org.example.sudoku;

-

* **import** android.content.Context;
* **import** android.media.MediaPlayer;

5

* **public class** Music {
* **private static** MediaPlayer mp = **null**;

-

* /\*\* Stop old song and start new one \*/

10 **public static void** play(Context context, **int** resource) {

* stop(context);
* mp = MediaPlayer.create(context, resource);
* mp.setLooping(**true**);
* mp.start();

15 }

-

* /\*\* Stop the music \*/
* **public static void** stop(Context context) {
* **if** (mp != **null**) {

20 mp.stop();

* mp.release();
* mp = **null**;

- }

- }

25 }

The play( ) method first calls the stop( ) method to halt whatever music is currently playing. Next, it creates a new MediaPlayer instance using MediaPlayer.create( ), passing it a context and a resource ID.

After we have a player, we then set an option to make it repeat the music in a loop and then start it playing. The start( ) method comes back immediately.

The stop( ) method that begins on line 18 is simple. After a little defen- sive check to make sure we actually have a MediaPlayer to work with, we call its stop( ) and release( ) methods. The MediaPlayer.stop( ) method, strangely enough, stops the music. The release( ) method frees system resources associated with the player. Since those are native resources, we can’t wait until normal Java garbage collection reclaims them. Leav- ing out release( ) is a good way to make your program fail unexpectedly (not that this has ever happened to me, of course; I’m just saying *you* should keep that in mind).

Now comes the fun part—try playing Sudoku with these changes in place. Stress test it in every way you can imagine, such as switch- ing to different activities, pressing the Back button and the Home but- ton from different points in the game, starting the program when it’s already running at different points, rotating the display, and so forth. Proper life-cycle management is a pain sometimes, but your users will appreciate the effort.

# Adding Options to Sudoku

In Section 3.7, *Adding a Menu*, on page 64, we used the onCreateOption- sMenu( ) method to add a menu containing one item to the main Sudoku screen. When the user presses the Menu key and selects the Settings... item, the code starts the Prefs activity, which lets the user change the options for the game. Because Prefs extends PreferenceActivity, the values for the settings are stored in the program’s preferences area, but orig- inally we didn’t do anything with them. Now we’re going to implement them.

**Sudoku Trivia**

There are 6,670,903,752,021,072,936,960 possible classic Sudoku solution grids. If you eliminate duplicates that are just rotations, reflections, or relabelings of each other, you’re left with “only” 5,472,730,538 solutions.

First let’s modify the Prefs class to add a couple of getter methods that retrieve the current values of our two options. Here’s the new definition:

Download **Sudokuv4/src/org/example/sudoku/Prefs.java**

**package** org.example.sudoku;

**import** android.content.Context;

**import** android.os.Bundle;

**import** android.preference.PreferenceActivity;

**import** android.preference.PreferenceManager;

**public class** Prefs **extends** PreferenceActivity {

// Option names and default values

**private static final** String OPT\_MUSIC = "music" ; **private static final boolean** OPT\_MUSIC\_DEF = **true**; **private static final** String OPT\_HINTS = "hints" ; **private static final boolean** OPT\_HINTS\_DEF = **true**;

@Override

**protected void** onCreate(Bundle savedInstanceState) { **super**.onCreate(savedInstanceState); addPreferencesFromResource(R.xml.settings);

}

/\*\* Get the current value of the music option \*/

**public static boolean** getMusic(Context context) {

**return** PreferenceManager.getDefaultSharedPreferences(context)

.getBoolean(OPT\_MUSIC, OPT\_MUSIC\_DEF);

}

/\*\* Get the current value of the hints option \*/

**public static boolean** getHints(Context context) {

**return** PreferenceManager.getDefaultSharedPreferences(context)

.getBoolean(OPT\_HINTS, OPT\_HINTS\_DEF);

}

}

Be careful that the option keys (music and hints) match the keys used in res/xml/settings.xml.

Music.play( ) has to be modified to check for the music preference:

Download **Sudokuv4/src/org/example/sudoku/Music.java**

**public static void** play(Context context, **int** resource) { stop(context);

// Start music only if not disabled in preferences

**if** (Prefs.getMusic(context)) {

mp = MediaPlayer.create(context, resource); mp.setLooping(**true**);

mp.start();

}

}

And PuzzleView.onDraw( ) also needs to be modified to check for the hints preference:

Download **Sudokuv4/src/org/example/sudoku/PuzzleView.java**

**if** (Prefs.getHints(getContext())) {

// Draw the hints...

}

If getHints( ) returns true, we draw the highlights for the hints, as shown in Figure 4.6, on page 92. Otherwise, we just skip that part.

Next I’ll show you how to use the preferences API to store things other than just options.

# Continuing an Old Game

At any time the player can decide to quit playing our Sudoku game and go do something else. Maybe their boss walked in, or they got a phone call or a notification of an important appointment. Whatever the reason, we want to allow the player to come back later and continue where they left off.

First we need to save the current state of the puzzle somewhere. The preferences API can be used for more than just options; it can store any small stand-alone bits of information that go with your program. In this case, the state of the puzzle can be saved as a string of eighty- one characters, one for each tile.

In the Game class, we’ll start by defining a couple of constants—one for the puzzle data key and one for a flag to tell us to continue the previous game rather than start a new one.

Download **Sudokuv4/src/org/example/sudoku/Game.java**

**private static final** String PREF\_PUZZLE = "puzzle" ;

**protected static final int** DIFFICULTY\_CONTINUE = -1;

Next we need to save the current puzzle whenever the game is paused. See Section 2.2, *It’s Alive!*, on page 35 for a description of onPause( ) and the other life-cycle methods.

Download **Sudokuv4/src/org/example/sudoku/Game.java**

@Override

**protected void** onPause() { **super**.onPause(); Log.d(TAG, "onPause" ); Music.stop(**this**);

// Save the current puzzle getPreferences(MODE\_PRIVATE).edit().putString(PREF\_PUZZLE,

toPuzzleString(puzzle)).commit();

}

Now the puzzle is saved, but how do we read the saved data? Remember that when the game is started, the getPuzzle( ) method is called, and the difficulty level is passed in. We’ll use that for continuing as well.

Download **Sudokuv4/src/org/example/sudoku/Game.java**

**private int**[] getPuzzle(**int** diff) { String puz;

**switch** (diff) {

**case** DIFFICULTY\_CONTINUE:

puz = getPreferences(MODE\_PRIVATE).getString(PREF\_PUZZLE, easyPuzzle);

**break**;

// ...

}

**return** fromPuzzleString(puz);

}

All we need to do is add a check for DIFFICULTY\_CONTINUE. If that is set, then instead of starting with a fresh puzzle, we read the one we stuffed into the preferences.

Next, we need to make the Continue button on the main screen (see Figure 3.4, on page 53) actually do something. Here is where we set that up.

Download **Sudokuv4/src/org/example/sudoku/Sudoku.java**

**public void** onClick(View v) {

**switch** (v.getId()) {

**case** R.id.continue\_button: startGame(Game.DIFFICULTY\_CONTINUE); **break**;

// ...

}

}

We added a case in Sudoku.onClick( ) to call startGame( ) when the Con- tinue button is pressed, passing it DIFFICULTY\_CONTINUE. startGame( ) passes the difficulty to the Game activity, and Game.onCreate( ) calls Intent.getIntExtra( ) to read the difficulty and passes that to getPuzzle( ) (you can see the code for that in Section 4.2, *Starting the Game*, on page 78).

There’s one more thing to do: restore from our saved game when our activity goes away and comes back on its own (such as if another activ- ity is started and then the user comes back to the Game activity). This modification to the Game.onCreate( ) method will take care of that:

Download **Sudokuv4/src/org/example/sudoku/Game.java**

@Override

**protected void** onCreate(Bundle savedInstanceState) {

// ...

// If the activity is restarted, do a continue next time getIntent().putExtra(KEY\_DIFFICULTY, DIFFICULTY\_CONTINUE);

}

That pretty much covers it for preferences. Next let’s look at saving instance state.

# Remembering the Current Position

If you change the screen orientation while Sudoku is running, you’ll notice that it forgets where its cursor is. That’s because we use a cus- tom PuzzleView view. Normal Android views save their view state auto- matically, but since we made our own, we don’t get that for free.

Unlike persistent state, instance state is not permanent. It lives in a Bundle class on Android’s application stack. Instance state is intended to be used for small bits of information such as cursor positions.

Here’s what we have to do to implement it:

Download **Sudokuv4/src/org/example/sudoku/PuzzleView.java**

Line 1 **import** android.os.Bundle;

* **import** android.os.Parcelable;

-

* **public class** PuzzleView **extends** View {

5 **private static final** String SELX = "selX" ;

* **private static final** String SELY = "selY" ;
* **private static final** String VIEW\_STATE = "viewState" ;
* **private static final int** ID = 42;

-

10 **public** PuzzleView(Context context) {

- // ...

* setId(ID);

- }

-

15 @Override

* **protected** Parcelable onSaveInstanceState() {
* Parcelable p = **super**.onSaveInstanceState();
* Log.d(TAG, "onSaveInstanceState" );
* Bundle bundle = **new** Bundle();

20 bundle.putInt(SELX, selX);

* bundle.putInt(SELY, selY);
* bundle.putParcelable(VIEW\_STATE, p);
* **return** bundle;

- }

25 @Override

* **protected void** onRestoreInstanceState(Parcelable state) {
* Log.d(TAG, "onRestoreInstanceState" );
* Bundle bundle = (Bundle) state;
* select(bundle.getInt(SELX), bundle.getInt(SELY));

30 **super**.onRestoreInstanceState(bundle.getParcelable(VIEW\_STATE));

* **return**;

- }

- // ...

- }

On line 5, we define some constants for keys to save and restore the cursor position. We need to save both our own x and y positions, plus any state needed by the underlying View class.

As part of Activity.onSaveInstanceState( ) processing, Android will walk down the view hierarchy and call View.onSaveInstanceState( ) on every view it finds that has an ID. The same thing happens for onRestoreIn- stanceState( ). Normally, this ID would come from XML, but since Puzzle- View was created in code, we need to set it ourselves. We make up an arbitrary number on line 8 (any value will do as long as it’s positive) and then use the setId( ) method to assign it on line 12.

The onSaveInstanceState( ) method is defined on line 16. We call the superclass to get its state, and then we save ours and theirs in a Bundle. Failing to call the superclass will result in a runtime error.

Later, onRestoreInstanceState( ) (line 26) will be called to tease out the information we saved. We get our own x and y positions from the Bundle, and then we call the superclass to let it get whatever it needs. After making these changes, the cursor will be remembered by PuzzleView, just like any other Android view.

Next let’s look at keeping data in plain old files.